Deriving Abstract Transfer Functions for Analyzing Embedded Software

John Regehr
School of Computing, University of Utah
regehr@cs.utah.edu

Usit Duongsaa
Microsoft
usitdu@microsoft.com

Abstract

This paper addresses the problem of creating abstract transfer functions supporting dataflow analyses. Writing these functions by hand is problematic: transfer functions are difficult to understand, difficult to make precise, and difficult to debug. Bugs in transfer functions are particularly serious since they defeat the soundness of any program analysis running on top of them. Furthermore, implementing transfer functions by hand is wasteful because the resulting code is often difficult to reuse in new analyzers and to analyze new languages. We have developed algorithms and tools for deriving transfer functions for the bitwise and unsigned interval abstract domains. The interval domain is standard; in the bitwise domain, values are vectors of three-valued bits. For both domains, important challenges are to derive transfer functions that are sound in the presence of integer overflow, and to derive precise transfer functions for operations whose semantics are a mismatch for the domain (i.e., bit-vector operations in the interval domain and arithmetic operations in the bitwise domain). We can derive transfer functions, and execute them, in time linear in the bitwidth of the operands. These functions are maximally precise in most cases. Our generated transfer functions are parameterized by a bitwidth and are independent of the language being analyzed, and also of the language in which the analyzer is written. Currently, we generate interval and bitwise transfer functions in C and OCaml for analyzing C source code, ARM object code, and AVR object code. We evaluate our derived functions by using them in an interprocedural dataflow analyzer.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Algorithms, Reliability, Languages, Verification

Keywords Static Analysis, Abstract Interpretation, Transfer Functions, Embedded Software

1. Introduction

The bottom layer of a dataflow analyzer is a collection of *transfer functions* that soundly abstract the behavior of low-level program operations in order to derive facts about the program that are true across all possible executions. Typically, transfer functions are implemented by hand and are specific to a particular abstract domain and a particular programming language being analyzed. Furthermore, these functions are embedded in, and are specific to, a particular program analyzer. Each transfer function must be correct (returning a conservative estimate of the effect of an operation on the program state), precise (losing as little information as possible), and as efficient as possible. Our experience is that meeting these goals is difficult, a view that is backed up by anecdotes we have heard from a number of other groups who have implemented sophisticated dataflow engines.

Transfer functions operate on *abstract values*, each of which corresponds to some set of concrete data values. Operations on sets of values are difficult to understand. Tedious but important tasks include dealing with integer overflow, implementing abstract effects on a processor's condition codes, and creating abstract versions of operations that do not match the domain, such as a bitwise "add" or an interval "xor."

The contribution of this paper is a collection of algorithms, and tools implementing these algorithms, that represent a step towards solving all of these problems. Specifically, we address the problem of automatically deriving precise and efficient abstract transfer functions for ALU operations. We have developed methods to derive these transfer functions with the following properties:

- We support the bitwise and unsigned interval domains. The interval domain is standard and in the bitwise domain, abstract values are vectors of three-valued bits (i.e., each bit is true, false, or unknown).
- Derivation is very efficient.
- The performance of the derived transfer functions is linear in the bitwidth of the target architecture.
- The derived transfer functions are maximally precise in most cases.
- The derived functions can be used to analyze different programming languages. Currently we support AVR object code, ARM object code, and C source code.
- The derived functions can be used in program analyzers written in different programming languages. Currently we generate transfer functions in C and OCaml.

The most closely related previous work is our Hoist project [13], which treated concrete machine operations as black boxes and lifted them into abstract domains using brute force algorithms. This minimized human effort and worked well for small embedded architectures, but did not scale to 16-, 32-, or 64-bit processors. The methods described in this paper are scalable, but this scalability has a cost in developer effort: a new set of algorithms has to be developed for each new abstract domain that is supported.

This paper is organized as follows. Section 2 provides background on abstract interpretation, on the two domains that we use,

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.

UU IR Author Manuscript

UU IR Author Manuscript

and on previous research for creating abstract transfer functions. Sections 3 and 4 respectively present our new algorithms for deriving abstract transfer functions for the bitwise and interval domains. In Section 5 we describe our tools that implement these algorithms and then in Section 6 we evaluate their precision and performance. Section 7 presents some discussion on deriving transfer functions and we conclude in Section 8.

2. Background

Dataflow analysis is a broadly useful technology for verifying, validating, and optimizing embedded software. For example:

- Lundqvist and Stenström [9] developed a worst-case execution time analyzer based on path-sensitive constant propagation.
- We bounded stack memory usage for interrupt-driven programs using a context-sensitive implementation of the bitwise domain [14].
- De Bus et al. [3] used constant propagation to optimize ARM binaries.

Many other applications of these techniques can be found in the literature.

2.1 Abstract interpretation

Abstract interpretation [2] is a framework that supports formal, modular reasoning about static program analysis. By manipulating abstract values, which represent sets of concrete values, an abstract interpreter can compute the properties of many program executions using relatively few analysis steps. For example, rather than separately analyzing the behavior of an embedded system for each of the possible values returned by a temperature sensor, an abstract interpreter would simply analyze the case where the sensor returns a single abstract value representing the set of all possible temperature inputs. Abstract interpretations deliberately make approximations to avoid undecidability and to achieve reasonable space and time efficiency.

Abstract domains Abstract values belong to *domains*, or partially ordered lattices of abstract values, where each abstract value x corresponds to a unique set of concrete values $\gamma(x)$ and, conversely, each set of concrete values Y corresponds to an abstract value $\alpha(Y)$. Although concrete domains are commonly taken to be infinite, e.g., $\mathbb{Z}$ or $\Re$, we observe that in practice, computer programs manipulate values the come from finite sets: each value must fit into some number of bits. The semantics of concrete operations come from the semantics of the processor or programming language for which we are deriving transfer functions.

The smallest element of a lattice $\perp$ represents a complete lack of information; its concretization is the set of all possible values. The partial order of a lattice is defined by:

$$x \sqsubseteq y \stackrel{\text{def}}{=} \gamma(x) \supseteq \gamma(y)$$

In other words, smaller abstract values represent larger sets of concrete values, and are consequently less precise estimations of the contents of a storage location. The greatest lower bound operation $\sqcap$ is the largest (most precise) abstract value such that:

$$\gamma(x \sqcap y) \supseteq \gamma(x) \cup \gamma(y)$$

In other words, the concretization of the greatest lower bound of two values must be a superset of the union of the concretization of the individual values. We refer to $\sqcap$ as the *merge* operator for two abstract values; it is used to create a safe and precise estimate of the state of the machine when two control-flow paths are merged, for example after analyzing both branches of an if-then-else construct.

In this paper an "imprecise" result is technically correct but has lost some information. It is important to distinguish between the different kinds of imprecision that occur during abstract interpretation. First, an abstract value may be imprecise because the abstract domain cannot represent a given concrete set. For example, consider a program where a certain storage location contains only the values "4" and "6" in all executions. A constant propagation analysis must conclude that the storage location contains the value $\perp$, because this domain is inherently not expressive enough to return a more precise result. Second, an abstract value may be imprecise because of approximations made in the implementation of an abstract interpreter. For example, assume that two different interval domain analyzers for the program above respectively estimate the storage location to contain $[4..6]$ and $[0..6]$. Both results are correct but only $[4..6]$ is maximally precise, given the constraints of the interval domain. This paper is about avoiding this second kind of imprecision. All other things being equal a more precise analysis is preferable, but usually precision is gained at the expense of memory and CPU consumption at analysis time.

Abstract operations The focus of this paper is on deriving an abstract operation f' for every ALU operation f that is provided by a given processor architecture or high-level language. An abstract operation must satisfy:

$$\alpha(\{f(y)|y \in \gamma(x)\}) \subseteq f'(a)$$

To understand this equation, consider an abstract value x. We can apply the concrete function f to each member of the set of concrete values represented by x. The set of concrete values so obtained must be a subset of the concretization of the abstract value returned by applying the abstract function f' to x.

The trivial abstract function, which always returns $\perp$, is correct—but useless. The challenge, then, is to obtain a more precise abstract function. In this paper, we use $f^\sharp$ to denote the most precise abstract version of a concrete function f for a given domain. The maximally precise abstract function can be computed by:

$$\begin{array}{l} f^\sharp(a) \stackrel{\text{def}}{=} \alpha(f(c_1)) \sqcap \ldots \sqcap \alpha(f(c_m)) \\ \text{where } \{c_1, \ldots, c_m\} = \gamma(a) \end{array} \quad (1)$$

That is: concretize the abstract value that is the argument to the function, apply the corresponding concrete function to each concrete value, and then merge the concrete results together to form an abstract value. The maximal precision follows from the fact that we are merging together only values that need to be merged, and the facts that $\sqcap$ is commutative, associative, and maximally precise.

The straightforward computation of $f^\sharp$ for a binary operator has runtime quadratic in the worst-case size of the concretization set, which is itself usually exponential in the bitwidth of the operands. Thus, simply evaluating Equation 1 inside a program analyzer is far too inefficient, even for 8-bit architectures. The primary contribution of this paper is a collection of algorithms for deriving abstract transfer functions such that both deriving and executing the functions requires linear time in the bitwidth of the operands.

2.2 Bitwise domain

The bitwise abstract domain is a ternary logic in which each bit either has a concrete value or is unknown. Formally, each bit has a value from the set $\{0, 1, \perp\}$ and the concretization function is:

$$\begin{array}{rcl} \gamma(0) & = & \{0\} \\ \gamma(1) & = & \{1\} \\ \gamma(\perp) & = & \{0, 1\} \end{array}$$

Consequently, the merge function for bits is:

$$a \sqcap_{bit} b \stackrel{\text{def}}{=} \begin{cases} a & \text{if} \quad a = b \\ \perp & \text{if} \quad a \neq b \end{cases} \quad (2)$$

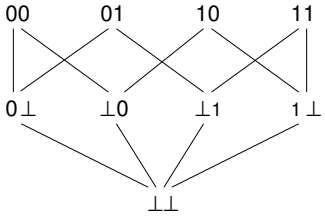

Figure 1. The bitwise lattice for $n = 2$. In the general case this lattice has height $n + 1$ and contains 3^n elements.

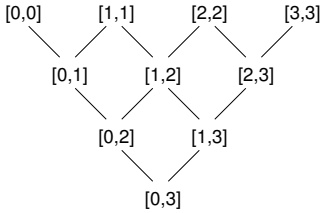

Figure 2. The unsigned interval lattice for $n = 2$. In the general case this lattice has height 2^n and contains $2^{n-1}(2^n + 1)$ elements.

An abstract word is the composition of multiple bits. We use n to denote the number of bits in a native machine word. The merge function for words is simply:

$$a \sqcap_{bits} b \stackrel{\text{def}}{=} (a_0 \sqcap_{bit} b_0, a_1 \sqcap_{bit} b_1, \ldots, a_{n-1} \sqcap_{bit} b_{n-1})$$

The bitwise lattice, shown for two bits in Figure 1, is useful for reasoning about partially unknown data that is manipulated using bitmasks and other logical operations. Our previous work on bounding the stack memory consumption of embedded software in the presence of interrupts [14] used the bitwise domain to estimate the status of the interrupt mask at each program point. The first use of the bitwise domain for analyzing software that we are aware of is Razdan and Smith's 1994 paper [12].

2.3 Interval domain

The interval domain [2] exploits the fact that although many storage locations contain values that change at run time, it is often the case that these values can be proven to occupy a sub-interval of the entire range of values that can be stored in a machine word. For example, it might be expected that a variable used to index an array of size i would never have a value outside the interval $[0..i - 1]$. In the interval domain, abstract values are tuples [low, high] where low $\leq$ high. The concretization function is:

$$\gamma([\text{lo}, \text{hi}]) \stackrel{\text{def}}{=} \{\text{lo}, \text{lo} + 1, \ldots, \text{hi}\}$$

Two intervals can be merged as follows:

$$[a_{lo}, a_{hi}] \sqcap_{int} [b_{lo}, b_{hi}] \stackrel{\text{def}}{=} [\min(a_{lo}, b_{lo}), \max(a_{hi}, b_{hi})] \quad (3)$$

The interval lattice, shown for two-bit unsigned integers in Figure 2, is best used to model arithmetic operations. It has been used as part of a strategy for eliminating array bounds checks [7], for bounding worst-case execution time [4], and for synthesizing optimized hardware by statically showing that the high-order bits of certain variables were constant at run time [17].

2.4 Ways to create transfer functions

Figure 3 compares three different ways of creating abstract transfer functions. This section outlines the problems with the completely manual and completely automated approaches.

Completely automated Our previous work on Hoist [13] used brute-force to lift concrete operations into abstract domains. Because Hoist treated both domains and program operations as black boxes, it required almost no input from human developers. However, lacking a semantic representation of domains or program operations, Hoist was forced to use unscalable algorithms. It worked well for small embedded processors but there is no obvious way to make it scale.

Completely by hand Transfer functions for simple abstract domains, such as null-pointer analysis and constant propagation, are easy to create by hand. On the other hand, elaborate domains such as intervals and bitwise values are not easy to implement, especially if both efficiency and precision are required. Consider an `add` instruction that has the following assembly language representation:

```
add dst, src
```

In a CPU simulator this might be implemented as:

```
reg[dst] = (reg[src]+reg[dst]) % (MAXUINT+1);
```

in addition to some code updating the condition code flags. We assume that the processor running the simulator has a larger word size than the processor being emulated, and hence `MAXUINT+1` does not overflow.

An abstract `add` in the unsigned interval domain is complicated slightly by the case analysis necessitated by the potential for the low and high ends of the result interval to independently wrap around:

```
lo = reg[src].lo + reg[dst].lo;
hi = reg[src].hi + reg[dst].hi;
if ((lo > MAXUINT) ^ (hi > MAXUINT)) {
  reg[dst].lo = 0;
  reg[dst].hi = MAXUINT;
} else {
  reg[dst].lo = lo % (MAXUINT+1);
  reg[dst].hi = hi % (MAXUINT+1);
}
```

Computing the abstract condition codes is also more difficult than it is in the concrete case, and both result and condition codes are even more painful for the signed interval domain where there are more ways to wrap around.

A bitwise abstract `add` is much trickier, and it is in situations such as this where implementers often resort to crude approximations. For example, a first cut at the bitwise `add` might return an entirely unknown result if any bit in any input is unknown. A better approximation is to return a result with m known bits if the bottom m bits of both arguments are known. For example, if bits 0–3 in both arguments are known, then the `add` functions normally in this range and returns $\perp$ for bits 4 and higher. On the other hand, if bits in position 4 are the only unknown bits in the inputs, and if the bits in position 5 of both inputs are zeros, then any possible carry out of position 4 will be absorbed, and the `add` can function normally again in bits 6 through $n - 1$ as if there were no unknown bits. Further improvements along these lines are possible but unattractive—the general case where known and unknown bits are freely mixed is difficult to reason about, as is the analogous case of computing the `xor` operation precisely for arbitrary interval values. In practice, developing a sufficiently good approximation for each machine-level operation is a laborious and error-prone process requiring refinement of approximations when the analysis returns imprecise results. Developing maximally precise functions without resorting to brute force is even more challenging.

Other previous work There has been very little work on generating transfer functions automatically, compared to the vast literature

derivation method	implementation effort	derivation efficiency	analysis efficiency	correctness	precision	scalability to large bitwidths
by hand	poor	N/A	very good	typically poor	typically poor	very good
this work	good	very good	acceptable	excellent	excellent	very good
Hoist [13]	very good	poor	acceptable	excellent	excellent	poor

Figure 3. Comparison of three different ways to obtain abstract transfer functions. The present work occupies a middle ground between two extremes: zero tool support, and Hoist's complete (but unscalable) tool support.

on dataflow analysis and abstract interpretation in general. Yorsh et al. [19] represent abstract values symbolically and manipulate them using a theorem prover. They generate a sequence of approximations to the answer, using counterexamples to find weaknesses of the current approximation. It is not really possible to directly compare this work with ours, as the underlying abstract domains are very different: Yorsh et al. focus on transfer functions for shape analysis. Rice et al. [16] derive transfer functions given dataflow fact schemas. The domains that they attack are traditional compiler domains such as constant propagation and pointer analysis, and it is not clear that their method could be extended to handle value propagation domains like intervals.

2.5 Notation used in this paper

In Boolean formulas we use the conventions that $+$ is Boolean-or, $\oplus$ is Boolean-xor, Boolean-and is implicit in adjacent terms, and Boolean complement is signified by a line over a term. In formulas we indicate low-level program operations in the concrete domain using a sans-serif font, e.g., xor. Abstract transfer functions in the bitwise and interval domains are indicated with subscripts, e.g., $\mathsf{xor}_{\mathsf{bit}}$ and $\mathsf{xor}_{\mathsf{int}}$ for bit-vector exclusive-or in the bitwise and interval domains, respectively. When interval abstract values appear in formulas and in pseudocode, we will use i_{lo} and i_{hi} to extract the lower and upper bounds, respectively, of interval i. Finally, when there is no danger of ambiguity we will abuse the notation slightly by interchangeably using "0" and "false," and "1" and "true."

3. Algorithms for the Bitwise Domain

This section presents our algorithms for deriving abstract transfer functions in the bitwise domain.

3.1 A restricted model of operations

Most ALU operations have a *ripple carry* structure. A ripple-carry operation is one where bit i of the output is a function of bit i in each of the inputs as well as a carry bit from position $i-1$. Bit-vector operations (other than right-shift) are trivially ripple-carry, as are all arithmetic operations other than multiply and divide. Multiply and divide are not heavily used in the embedded programs that we are interested in.

Let a and b be the inputs to a binary operation f that we wish to lift into the bitwise domain (we do not cover unary operations here—they are a straightforward degenerate case of the binary operations). Let r be the result of the operation, so that:

$$r = f(a, b)$$

Let c be a carry word that is an intermediate result of the operation. We identify the ith bits of a, b, c, and r respectively as a_i, b_i, c_i, and r_i.

A ripple-carry operation can be completely specified by Boolean functions x and y:

$$\begin{aligned} r_i &= x(a_i, b_i, c_i) \\ c_i &= y(a_{i-1}, b_{i-1}, c_{i-1}) \end{aligned}$$

in addition to a constant c_0 that determines the least significant bit of the carry word. We call x and y the *characteristic functions* for an operation. The value of f is then simply:

$$f(a, b) = \sum_{i=0}^{n-1} r_i 2^i$$

In ripple-carry form, integer addition is:

$$\begin{aligned} x(a, b, c) &= a \oplus b \oplus c \\ y(a, b, c) &= ab + ac + bc \\ c_0 &= 0 \end{aligned}$$

Without loss of generality we henceforth assume that $c_0 = 0$, since for any set of characteristic functions where $c_0 = 1$, complementing the c term in x and y gives functions with the same effect where $c_0 = 0$.

Given this limited model for machine operations, it is feasible to compute the characteristic functions for each operation through limited use of brute force. Since there are only 2^{16} possible combinations of functions x and y, we can simply try each one until we find the pair corresponding to the desired operation. In practice, this can be optimized further by a two-pass search where the first pass limits our search space to only those pairs that match the desired operation at position zero. Given our earlier assumption that $c_0 = 0$, this first pass can be performed quickly, significantly reducing the search space for the brute-force second pass.

3.2 Lifting Boolean functions into the bitwise domain

Reps et al. [15] showed how to compute a maximally precise three-valued version of a two-valued function. This requires exponential time and is analogous to a straightforward evaluation of Equation 1. In practice this is prohibitive for operations on 16 or more bits. On the other hand, characteristic functions are small enough that it is possible to rapidly create maximally precise three-valued functions. We wrote a solver that does this using brute force. The three-valued characteristic function can then be run in a loop to create the overall transfer function.

3.3 Multiplication

Multiplication does not fit the ripple-carry model. Nevertheless, we implemented a not-maximally-precise multiplication transfer function by exploiting our existing (maximally precise) shift and add transfer functions. In pseudocode form:

```
Bitwise Multiply (Bitwise in1, Bitwise in2)
  Bitwise Sum = IntToBitwise (0)
  For int i=0 to N-1
    Bitwise Product =
      MultiplyBit (in1, getBit(in2, i))
    Sum = Add (Sum, shiftLeft (Product, i))
  return Sum
```

In this example `Add` and `shiftLeft` are the existing transfer functions and `MultiplyBit` is a simple function that kills all "1" bits in its first argument if its second argument is bottom. We quantify the precision of this function in Section 6.

UU IR Author Manuscript

3.4 Condition codes

The model we have described so far does not consider condition codes. Of course, most computer architectures support both input and output condition codes. A common flag is *carry*, which captures the extra bit that would ordinarily be lost when adding or subtracting two n-bit quantities. The add-with-carry instruction, for example, uses the carry flag both as input and output, allowing two n-bit operations to be chained together into a $2n$-bit operation.

Our bitwise domain algorithms support carry as in input by dropping our earlier assumption that c_0 is always zero and instead letting it be a function of the incoming condition codes. Thus, the algorithms that we have already described suffice to handle carry inputs. Arbitrary condition code inputs would be difficult to handle, but no architecture that we have looked at uses any bit other than carry as input.

To derive transfer functions for condition codes that are outputs of instructions, we require that a developer provide a symbolic version of the condition code computation using a collection of building blocks that we have provided. For example, the following code describes three flags for the Atmel AVR processor:

```
Z = new FFResultEqualsZero();
H = new FFCarryOutBit (3,1);
V = new FFColumn (new FFColumn (7, 1, 1, 0),
                  new FFColumn (7, 0, 0, 1));
```

The Z (zero) flag uses a built-in primitive to check if the result of an instruction is zero. The H (half-carry) flag checks if the carry out of bit three is one. Finally, the V (two's complement overflow) flag is true if the tuple (top bit of operand one, top bit of operand two, top bit of the propagated carry) matches either the pattern $(1, 1, 0)$ or $(0, 0, 1)$. These computations are straightforward translations of information from the AVR instruction set reference.

For most flags, we lift the concrete functions for condition code flags into the bitwise domain in the same way we lift the characteristic functions of each operation. The Z flag is special in that it depends on values from every column as opposed to other flags which can be determined by looking at a single column of the operands and/or result. To support the Z flag we implemented a small custom SAT solver that gains efficiency by exploiting our ripple-carry model of instructions.

4. Algorithms for the Interval Domain

This section describes our algorithms for lifting concrete operations into the unsigned interval domain. The interval domain is more difficult than the bitwise domain in two ways. First, its lattice is larger, containing on the order of 4^n elements, as opposed to 3^n for the bitwise domain. Second, bits in the interval domain are not independent, as they are in the bitwise domain, making a divide-and-conquer approach to generating transfer functions more difficult.

Abstract arithmetic operations in the interval domain are relatively simple: they can be implemented by operating only on the upper and lower bounds of the input intervals, as illustrated in Section 2.4. The challenge is implementing precise and efficient abstract versions of bit-vector operations: these cannot be implemented solely in terms of their lower and upper bounds. Consider for example the interval version of the bit-vector "and" function:

$$[7..12]\ \mathsf{and}_{\mathsf{int}}[10..17] = [0..12]$$

No straightforward function of the bounds on the input intervals leads to the output intervals. Rather, the lower bound of the result, 0, comes from anding together 7 from the first input interval and 16 from the second. The upper bound of 12 comes from anding together 12 from the first input interval and 15 from the second.

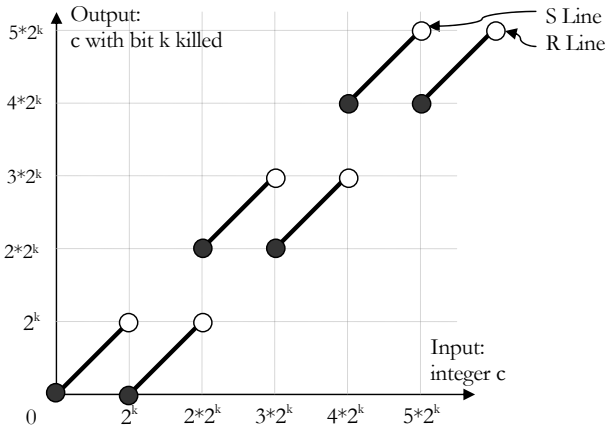

Figure 4. Visualizing the $\mathsf{KillBit}(c, k)$ function, which subtracts 2^k from an integer only if its kth bit is set, in which case we say that c lies on the "reduced" (R) line rather than the "standard" (S) line

Considering all such pairs is basically an implementation of Equation 1, which is unacceptable since it has performance exponential in the bitwidth of the operands. Subsequent subsections present a series of building blocks that eventually support derivation of maximally precise interval transfer functions for bit-vector operations.

4.1 The $\mathsf{not}_{\mathsf{int}}$ function

The bit-vector complement operator flips each bit in a word. The interval transfer function for bit-complement, $\mathsf{not}_{\mathsf{int}}$, works as follows. First, the complements of the upper and lower bounds of the input interval are computed. Second, these two numbers are swapped to form the lower and upper bounds of the result interval, respectively. This works because the bit-vector complement function for unsigned integers is monotonically decreasing.

4.2 The $\mathsf{KillBit}_{\mathsf{int}}$ function

$\mathsf{KillBit}_{\mathsf{int}}$ is a function from intervals to intervals that we use as a building block for other interval transfer functions. It is:

$$\mathsf{KillBit}_{\mathsf{int}}(a, k) \stackrel{\text{def}}{=} \alpha(\mathsf{KillBit}(c_1, k)) \sqcap \ \ldots\ \sqcap \alpha(\mathsf{KillBit}(c_m, k))$$
$$\text{where } \{c_1, \ldots, c_m\} = \gamma(a)$$
$$\text{and } \mathsf{KillBit}(c, k) \stackrel{\text{def}}{=} \begin{cases} c & \text{if} \quad c \bmod 2^{k+1} < 2^k \\ c - 2^k & \text{otherwise} \end{cases}$$

In other words, the effect of $\mathsf{KillBit}_{\mathsf{int}}$ is to enumerate the concretization set of its interval argument, zero the kth bit of each resulting concrete value, and then merge the results back together into an abstract value. For example $\mathsf{KillBit}_{\mathsf{int}}([13..21], 4) = [0..15]$.

The naive implementation of $\mathsf{KillBit}_{\mathsf{int}}$ is exponential in the bitwidth of the argument interval. Our constant-time version of $\mathsf{KillBit}_{\mathsf{int}}$ is suggested by Figure 4, which illustrates the effect of the KillBit function. A case analysis is required to compute each of the bounds in the result interval.

Upper bound If the upper bound of the input interval lies on the S line, $\mathsf{KillBit}_{\mathsf{int}}$ is trivial: the upper bound of the output interval is just the upper bound of the input interval. On the other hand, if the upper bound of the input interval lies on the R line, things are more complicated. Observe graphically that if there is a point on the S line to the left of an R point, the S point is guaranteed to be vertically higher (i.e., numerically greater once transformed). It is therefore necessary to check for the predecessor of the upper end

point. The predecessor is the right-most S point. If the input interval contains no S point, then the original upper end will be used. Pred is defined as follows:

$$\text{pred}(c,i) \stackrel{\text{def}}{=} \begin{cases} d & \text{if} \quad d \geq i_{lo} \\ c & \text{otherwise} \end{cases}$$

$$\text{where } d = c - (c \bmod 2^k) - 1$$

Lower bound If the lower bound of the input interval lies on the R line then the lower bound of the result interval is simply 2^k subtracted from the lower bound of the input interval. If the lower bound lies on the S line, things are more complicated in a way analogous to the upper bound lying on the R line, and can be computed be evaluating the "successor" function of the input lower bound:

$$\text{succ}(c,i) \stackrel{\text{def}}{=} \begin{cases} d & \text{if} \quad d \leq i_{hi} \\ c & \text{otherwise} \end{cases}$$

$$\text{where } d = c - (c \bmod 2^k) + 2^k$$

4.3 The $\mathsf{or}_{\mathsf{int}}$ function

Let r be $a \,\mathsf{or}_{\mathsf{int}}\, b$. A straightforward way to compute r using Equations 1 and 3 is:

$$r_{lo} = \min(x \,\mathsf{or}\, y) \text{ for all } x \in \gamma(a), y \in \gamma(b)$$

with r_{hi} defined analogously. Of course, implementations of abstract operations based on explicit concretization are too slow. For example, using this formula to compute $\bot \,\mathsf{or}_{\mathsf{int}}\, \bot$ for 32-bit integers requires on the order of 2^{64} operations.

Given $\mathsf{KillBit}_{\mathsf{int}}$, we can derive an efficient and maximally precise unsigned interval transfer function for bit-vector "or." We claim that

$$r_{lo} = \min \begin{cases} a_{lo} \,\mathsf{or}\, m_b \\ b_{lo} \,\mathsf{or}\, m_a \end{cases}$$

where m_a and m_b are numbers carefully chosen from intervals a and b. First we argue that this is true, and second we will show how to compute m_a and m_b.

The first step is to argue that the lower bound of the result is computable using either the lower bound of the first operand and the entire interval of the second, or else the lower bound of the second operand and the entire interval of the first, whichever is lower. The argument is as follows. Suppose a lower bound r is computed as $a \,\mathsf{or}\, b$ where a and b are numbers from the two operand intervals but neither is a lower bound. It can be proven that a new bound r' such that $r' \leq r$ can always be calculated from some numbers a' and b' such that $a' < a$ or $b' < b$.

Furthermore, we argue that a' and b' can be computed by simply subtracting one from either a or b, or both. Specifically, if either variable is odd, we subtract one from it, making it even. If both a and b are odd, we subtract one from each. Either way, the result is that both variables end up even. When we or them together, the result r' will be identical to r except its least significant bit will be zero, effectively making it one less than before, thus proving that a better lower bound exists.

The difficult case happens when both a and b are already even, i.e., the least-significant bit of each is zero. In this case we successively look at more significant bits until in one variable or the other we find a one bit. We then subtract one from that variable, or from both of them, if their right-most one bits are in the same position. Orring the resulting variables together will once again yield a better lower bound.

The second step is to compute m_a using the following algorithm:

```
int lowerBound (Interval a, Interval b)
  for int i=N-1 downto 0
    if isBitSet (b.lo, i)
      a = killBit_int (a, i)
  return a.lo
```

Computing m_b can be done by swapping all as with bs and vice-versa in the above algorithm. The logic behind the computation of m_a is as follows. Recall that we are trying to find the smallest value in the interval a that, once orred with b_{lo}, yields the smallest result possible. We find all bits of b_{lo} that are ones and kill the corresponding bits in the interval a. These bits are killed because they do not contribute to the result: they are already ones in b_{lo}, and the or operation only needs one side of the operand to be one. Once we kill these non-performing bits, we know that all remaining ones do contribute to the result, and orring any value from this interval with b_{lo} is equivalent to adding it to b_{lo}. Because we want the smallest result, we simply pick the lower bound of this interval.

4.4 The $\mathsf{and}_{\mathsf{int}}$ and $\mathsf{xor}_{\mathsf{int}}$ functions

Finally we are in a position to derive interval versions of the remaining bit-vector operations as follows:

$$\begin{aligned} a \,\mathsf{and}_{\mathsf{int}}\, b &\stackrel{\text{def}}{=} \mathsf{not}_{\mathsf{int}}(\mathsf{not}_{\mathsf{int}}(a) \,\mathsf{or}_{\mathsf{int}}\, \mathsf{not}_{\mathsf{int}}(b)) \\ a \,\mathsf{xor}_{\mathsf{int}}\, b &\stackrel{\text{def}}{=} (a \,\mathsf{and}_{\mathsf{int}}\, \mathsf{not}_{\mathsf{int}}(b)) \,\mathsf{or}_{\mathsf{int}}(\mathsf{not}_{\mathsf{int}}(a) \,\mathsf{and}_{\mathsf{int}}\, b) \end{aligned}$$

Both functions are correct and maximally precise, although more efficient versions could no doubt be created by deriving the functions from first principles. Although we have not proved maximal precision, our informal argument is that it follows from the maximal precision of the building blocks and from the fact that information is not "remembered" across multiple building-block operations. We have verified maximal precision and correctness of these operations using exhaustive testing for small bitwidths.

4.5 Condition codes

To support input condition codes, we need to detect the appropriate bit in the input flag word and adjust the lower and upper bounds of one of the operands accordingly. For all architectures that we are aware of, the only flag we need to worry about here is the carry flag. If the carry bit is set, both the lower and upper ends must be incremented, where as if the bit is unknown, we increment only the upper end of the interval to reflect the wider range of uncertainty.

The interval output condition codes are computed from the same symbolic representation of instructions that are used for the bitwise condition codes. However, we currently only support a subset of the codes that can be implemented maximally precisely. The Z (zero) flag is straightforward to implement: since zero lies on one end of the unsigned integer number line, we can easily determine whether an interval is exactly zero, may or may not be zero, or cannot be zero.

Another class of flags that we implemented is the one that, for the concrete domain, can be determined by examining only one bit of either operand or one bit of the result. The S (sign) flag found in most architectures is one example. For the two's complement encoding, the sign of a value can be determined by examining its most significant bit. For the interval domain, however, we need to employ the algorithm we used before. First we check the distance between the lower and upper bounds. If the distance is sufficiently far apart (greater than or equal to 2^k where k is the bit in question) then the bit must be unknown. Otherwise, we check whether the lower and upper end points are on the S or R line. If they fall on different lines, then clearly this bit is also unknown, as it is set during some parts of the line (the S part) but not the other. On the contrary, if they both fall on the S line then the bit is a known one, and a known zero if they both fall on the R line.

UU IR Author Manuscript

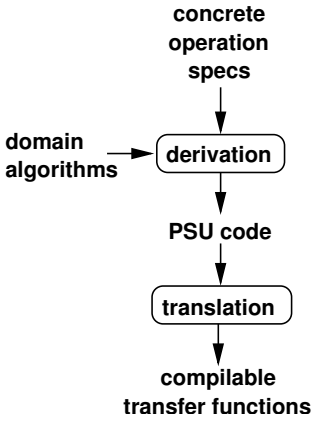

Figure 5. Deriving transfer functions

5. Implementation

We implemented the algorithms described in Sections 3 and 4 in a collection of tools totaling about 11,000 lines of C++ and 2,000 lines of OCaml. Figure 5 shows the flow of information in our implementation, which operates as follows:

1. The derivation algorithms are applied to the concrete operations, lifting them into the abstract domains. The resulting abstract transfer functions are independent of the bitwidth of the analyzed datatypes and they are in PSU: a simple and generic imperative language that we developed as an intermediate representation for transfer functions.
2. A back-end instantiates the PSU functions with specific bitwidths and translates them into a programming language.
3. The derived transfer functions are compiled and linked into a program analysis tool.

5.1 Analyzing multiple languages

We can currently generate transfer functions in the bitwise and interval domains for the following languages:

- AVR object code [1]
- ARM7 object code [8]
- C source code

Independence of the language being analyzed is accomplished using a specification language that is embedded in C++. A developer using this language writes C++ code instantiating objects that we have provided in order to build up a model of the behavior of each operation.

For example, the following code specifies the ARM or instruction:

```
void ArchArm::Or (const ConcrValue in1,
                  const ConcrValue in2,
                  const ConcrValue inCC,
                  ConcrValue& out1,
                  ConcrValue& outCC,
                  InstrDesc& desc)
{
  desc.name = "or";
  desc.op = new OpOr();
  ConcrValue mask = getArch().getMaxInt();
  out1 = (in1 | in2) & mask;
  useDefaultArmFlags(desc,"NZC");
}
```

This specification language is somewhat ad-hoc. In the long run we plan to exploit existing formal semantics for machine architectures, such as Lambda-RTL [11] or HOL's model of the ARM architecture [6].

5.2 Supporting analysis tools in multiple languages

Our tools emit transfer functions in PSU: a small imperative language that we designed to be strongly typed, easy to parse, and easy to translate into other languages. PSU supports expressions, conditionals, function calls, and for-loops. Values and operations in PSU are conceptually arbitrary precision; it is up to the PSU translator to choose appropriate types so that information is not lost. For example, interval transfer functions for n-bit integers require $n + 1$ bits of precision. Therefore, transfer functions for datatypes of 16 and fewer bits can use native 32-bit integers. Analyzing 32-bit integers requires 64-bit integers, and analyzing 64-bit integers requires access to a bignum package. Because many languages do not provide a uniform interface to native and non-native integer types, the PSU translator must be made aware of the different interfaces.

This fragment of PSU code is taken from the $\mathsf{or}_{\mathsf{bit}}$ function for the ARM architecture:

```
INT_VAR r;
ASSIGN r GETS (((((((b)BITWISE_OR(a)))BITWISE_AND
  (0xffffffff)))BITWISE_AND(rk)));
ASSIGN outCCk
  GETS ((((EXTRACT_KNOWN(inCC)))BITWISE_AND(0x1fffffff)));
ASSIGN outCCv
  GETS ((((EXTRACT_VALUE(inCC)))BITWISE_AND(0x1fffffff)));
IF (INT2BOOL(((dk)BITWISE_AND((1)SHIFT_LEFT
  ((ARCH_WORD_SIZE)MINUS(1))))))) BEGIN_BLOCK
  ASSIGN outCCk
    GETS ((((EXTRACT_KNOWN(outCC)))BITWISE_OR(0x20000000)));
END_BLOCK
```

Currently we translate PSU into C and OCaml. A Java backend would be straightforward to write, and would permit us, for example, to replace the hand-written transfer functions for the bitwise domain for the AVR architecture in Avrora [18] with our derived functions.

5.3 Supporting diagonal operations

The precision of many transfer functions can be improved when both operands come from the same physical location. For example, $x \,\mathsf{xor}_{\mathsf{int}}\, x = 0$ even if $x = \bot$ before the operation. The generic $\mathsf{xor}_{\mathsf{int}}$ transfer function has to return bottom in this case. We refer to operations where both inputs come from the same location as *diagonal* because their results can be found on the diagonal of the result table for the function. Support for diagonal operations is important when analyzing object code because, for example, many compilers use exclusive-or to clear variables rather than loading an immediate zero. Our tools automatically create a diagonal version of every transfer function that they emit.

6. Evaluation

This section evaluates our derived transfer functions.

6.1 Validation

We exhaustively tested our generated transfer functions for both correctness and precision whenever possible. To exhaustively test a transfer function, all possible combinations of abstract values are tested against a slow—but obviously correct—version of the transfer function that implements Equation 1.

To make exhaustive testing practical we had to artificially restrict the bitwidth of most transfer functions. Of course limited-bitwidth testing reduces our confidence in the test results. However, intuitively, it is difficult to see how a 32-bit interval xor could

UU IR Author Manuscript

fail when the 8-bit operation (which is derived from the same PSU code as the 32-bit version) succeeds. Even so, we also performed probabilistic testing of large-bitwidth operations. We did this by generating random abstract values whose concretization sets were small enough that Equation 1 could be evaluated in a reasonable amount of time.

Although exhaustive and probabilistic testing uncovered bugs early in our work, the final versions of our transfer functions are believed to be correct, and to be maximally precise except in cases where we have indicated that maximal precision was not achievable.

6.2 Microbenchmarks

We compared the precision and throughput of our derived transfer functions with brute-force transfer functions and also with two kinds of manually implemented transfer functions. First, those that we optimized for speed, and second, those that we optimized for precision. Because manual implementation is extremely tedious—the point of this work is to avoid it in the first place—only a few of these functions were implemented. The results are summarized in Figure 6. The derived transfer functions evaluated here were the ones for analyzing C code, and they were implemented in C.

These results confirm that automatically generated transfer functions are more precise than manually written ones. While the automatically generated functions are maximally precise, the manually written ones lose anywhere from 2.8 to 11.2 bits per operation, or as much as 48% of the information that could have been gained. If precision is an important goal, the automatically generated functions are clearly preferable. On the other hand, our automatically generated functions are also slower than those written by hand. The performance penalty ranges from 11% (comparing against 32-bit precision-optimized functions) to 500% (8-bit performance-optimized functions).

Our tool implementations do not attempt to generate optimized code, either during PSU generation, or in the translation of PSU code into C and OCaml. We optimized some of our generated transfer functions by hand and found that it was not difficult to speed them up by a factor of two. This benefit was attained by applying standard compiler optimizations that—for whatever reason—`gcc` was not applying, even at high optimization levels. It is likely that these optimizations could be automatically applied by our toolchain. We discuss two more aggressive ways to improve performance of the generated code in Section 7.

6.3 Macrobenchmarks

To evaluate our transfer functions in the context of actual program analyses, we ported them into an interprocedural dataflow analyzer that we have developed. Our analyzer is based on CIL [10] and is implemented in OCaml. It handles all of C and specifically targets embedded software, for example by supporting analysis of concurrent programs. Figure 7 uses several different metrics to compare interval and bitwise analyses based on our transfer functions to the well-known constant propagation analysis. *Dead code* is the percentage of statements in the original program that our analyzer can show to be unreachable. *Constant variables* refers to uses of variables that can be replaced with constants, as a percentage of the total number of static uses of variables in the program. *Bits known* is a domain-independent measure of analysis precision that tracks the number of bits in program variables that the analysis can show to be invariant across all executions. Finally, *analysis time* is just the time to run our analyzer, including the (often substantial) time taken by CIL's pointer analysis.

6.4 Ease of use

The main goal of our research is to reduce the work required to generate abstract transfer functions. Although we do not have quantitative results about this, we do have some anecdotal evidence. One of the authors estimates that it takes less than 10 minutes to write the specification for an instruction, and another 5–15 minutes to set up an automated tester. On the other hand, writing an abstract transfer function from scratch, debugging it, and tweaking it to improve precision and performance can take anything from a few minutes to several hours, depending on the complexity of the instruction, the desired level of precision, and the desired level of performance. Moreover, lingering errors in transfer functions can directly lead to overall unsound results. Similarly, lingering imprecision can easily lead to overall analysis failures because some critical piece of information remains unknown. Both kinds of failure are difficult to debug because they occur at such a low level.

We find abstract transfer functions to be hard to think about. Condition code flags and wraparound conditions are particularly tricky. In several instances, the authors of this paper spent up to half an hour arguing with each other about the correctness or precision of a manually implemented transfer function.

7. Discussion

Transfer functions comprise the bottom layer of a dataflow analyzer and are consequently independent of higher-level analysis design choices such as flow (in)sensitivity, path (in)sensitivity, and context (in)sensitivity. Consequently, our derived transfer functions should be broadly useful, and we believe that our approach can be straightforwardly applied to more programming languages, byte codes, and assembly languages.

On the other hand, it is not clear how to generalize our work to additional abstract domains. In fact, in our view the most dissatisfying thing about this work is that we were unable to create a unified model of ALU operations that generalized across many different abstract domains. In contrast, Hoist [13] did have such a unified model. However, we have achieved linear function derivation times compared to Hoist's exponential runtime.

In Section 6 we showed that hand-written transfer functions typically outperform our automatically derived functions. We have two ideas—that we leave for future work—that could greatly speed up the generated code:

1. Our generated functions are Boolean formulas that almost certainly contain some redundancy. Consider, for example, our implementation of $\mathsf{xor}_{\mathsf{int}}$ in Section 4.4—it clearly contains considerable low-level redundancy. Redundant subcomputations could be eliminated by creating a vector of binary decision diagrams (BDDs) for each transfer function, and then generating code that evaluates the BDDs instead of evaluating the function in its original form. This would eliminate redundant computations even, for example, across multiple condition code bits.
2. Our transfer functions fail to exploit the parallelism that all processors expose through bit-vector instructions. A technique such as SIMD within a register [5] could be used to recapture some of this available parallelism.

The work presented in this paper is just a first step. In the future we plan to derive transfer functions for additional abstract domains, and to develop proofs of correctness for our existing algorithms. We would also like to create a less ad-hoc specification language for concrete operations, or to reuse existing formal semantics for high-level languages, byte codes, and assembly languages.

UU IR Author Manuscript

operation	bits	domain	Performance (1000x ops/sec)				Average precision loss per op. (bits)			
			brute force	manual (speed)	manual (precision)	auto. gen.	brute force	manual (speed)	manual (precision)	auto. gen.
subtract	8	bitwise	2	3948	1549	882	0	3.9	2.8	0
subtract	32	bitwise	$\ll 1$	1641	356	321	0	11.2	6.5	0
xor	8	bitwise	2	4841	-	1943	0	0	-	0
xor	32	bitwise	$\ll 1$	4760	-	875	0	0	-	0
subtract	8	interval	<1	-	-	847	0	-	-	0
subtract	32	interval	$\ll 1$	-	-	818	0	-	-	0
xor	8	interval	<1	-	-	42	0	-	-	0
xor	32	interval	$\ll 1$	-	-	16	0	-	-	0
multiply	8	bitwise	2	-	-	4	0	-	-	4.2
multiply	32	bitwise	$\ll 1$	-	-	0.4	0	-	-	4.6

Figure 6. Transfer function microbenchmark results

benchmark	lines of C	domain	dead code (%)	constant variables (%)	bits known (%)	analysis time (sec.)
yacr2	4001	constant	5.51	0.71	4.96	1
		bitwise	5.51	0.71	6.43	10
		interval	5.51	0.71	5.40	5
bzip2	7565	constant	3.28	0.00	2.82	39
		bitwise	3.53	0.34	8.52	74
		interval	3.53	0.34	6.04	73
bc	7341	constant	2.86	0.10	0.21	85
		bitwise	2.86	0.10	4.26	132
		interval	2.86	0.10	4.06	120
kitchen	8466	constant	12.69	0.00	13.14	103
		bitwise	57.42	1.51	55.33	114
		interval	57.42	1.51	55.09	119

Figure 7. Whole-program static analysis results for the two derived domains, compared to the constant propagation domain

8. Conclusion

Precise and correct abstract transfer functions are an essential ingredient for static analyses, but they are difficult to implement. The paper has two main contributions. First, a collection of algorithms for lifting concrete program operations into the bitwise and unsigned interval abstract domains. The derived transfer functions are bitwidth-independent and in most cases they are maximally precise: they lose no precision beyond what they are forced to lose by the structure of the abstract domain. Our second main contribution is a collection of tools instantiating these algorithms in such a way that they can be easily used to analyze a new instruction set or programming language, and such that the resulting transfer functions are easy to automatically translate into a new programming language. The main idea is to amortize our work on algorithmically lifting concrete operations into abstract domains to the largest possible extent. We have shown that transfer functions for analyzing AVR object code, ARM object code, and C source code can be derived, and also that they can be mechanically translated into C and OCaml code that is ready to be linked into static analysis tools. We have shown that the derived transfer functions can be used in an actual program analysis tool.

9. Acknowledgments

We thank the reviewers for their helpful comments. This work is supported by National Science Foundation CAREER Award CNS-0448047.

References

[1] Atmel Corp. Atmel AVR 8-bit RISC family. `http://www.atmel.com/products/avr`.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symp. on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, January 1977.

[3] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. Link-time optimization of ARM binaries. In *Proc. of the 2004 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 211–220, Washington, DC, June 2004.

[4] Jakob Engblom, Andreas Ermedahl, Mikael Nolin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *Journal of Software Tool and Transfer Technology (STTT)*, 4(4):437–455, August 2003.

[5] Randall J. Fisher and Henry G. Dietz. Compiling for SIMD within a register. In *Proc. of the 11th Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 290–304, Chapel Hill, NC, 1998.

[6] Anthony Fox. Formal specification and verification of ARM6. In *Proc. of the 16th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 25–40, Rome, Italy, September 2003.

[7] John K. Gough and Herbert Klaeren. Eliminating range checks using static single assignment form. In *Proc. of the 19th Australian Computer Science Conf.*, Melbourne, Australia, January 1996.

[8] ARM Ltd. ARM7 32-bit RISC Family. `http://www.arm.com/products/CPUs/families/ARM7Family.html`.

[9] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, 17(2/3):183–207, November 1999.

[10] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, pages 213–228, Grenoble, France, April 2002.

[11] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *Proc. of the 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 176–192, Montreal, Canada, June 1998.

[12] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. of the 27th Intl. Symp. on Microarchitecture (MICRO)*, pages 172–180, San Jose, CA, November 1994.

[13] John Regehr and Alastair Reid. Hoist: A system for automatically deriving static analyzers for embedded systems. In *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[14] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, October 2003.

[15] Thomas Reps, Alexey Loginov, and Mooly Sagiv. Semantic minimization of 3-valued propositional formulae. In *Proc. of the 17th IEEE Symp. on Logic in Computer Science (LICS 2002)*, Copenhagen, Denmark, July 2002.

[16] Erika Rice, Sorin Lerner, and Craig Chambers. Automatically inferring sound dataflow functions from dataflow fact schemas. In *Proc. of the 4th Intl. Workshop on Compiler Optimization Meets Compiler Verification*, Edinburgh, UK, April 2005.

[17] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 108–120, Vancouver, Canada, June 2000.

[18] Ben L. Titzer, Daniel Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, Los Angeles, CA, April 2005.

[19] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proc. of the 10th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Barcelona, Spain, March 2004.

UU IR Author Manuscript